Experimental Evaluation of a Branch and Bound Algorithm for Computing Pathwidth and Directed Pathwidth

David Coudert, Dorian Mazauric, Nicolas Nisse

Experimental Evaluation of a Branch and Bound Algorithm for Computing Pathwidth and Directed Pathwidth*

David Coudert[1,2] Dorian Mazauric[1] Nicolas Nisse[1,2]

1. Inria, France
2. Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France

Abstract

Path-decompositions of graphs are an important ingredient of dynamic programming algorithms for solving efficiently many NP-hard problems. Therefore, computing the pathwidth and associated path-decomposition of graphs has both a theoretical and practical interest. In this paper, we design a Branch and Bound algorithm that computes the exact pathwidth of graphs and a corresponding path-decomposition. Our main contribution consists of several non-trivial techniques to reduce the size of the input graph (pre-processing) and to cut the exploration space during the search phase of the algorithm. We evaluate experimentally our algorithm by comparing it to existing algorithms of the literature. It appears from the simulations that our algorithm offers a significant gain with respect to previous work. In particular, it is able to compute the exact pathwidth of any graph with less than 60 nodes in a reasonable running-time ($\leq$ 10 minutes on a standard laptop). Moreover, our algorithm achieves good performance when used as a heuristic (i.e., when returning best result found within bounded time-limit). Our algorithm is not restricted to undirected graphs since it actually computes the directed pathwidth which generalizes the notion of pathwidth to digraphs.

1 Introduction

Because of their well known algorithmic interest, a lot of work has been devoted to the computation of *treewidth* and *tree-decompositions* of graphs [9, 19, 20, 50]. On the theoretical side, exact exponential algorithms [2, 12, 13, 34], Fixed Parameter Tractable (FPT) algorithms [8, 16] and approximation algorithms [11, 32, 48, 54] have been designed. Unfortunately, most of these algorithms are impractical for large graphs. For instance, the algorithms in [2, 34] are Branch and Bound algorithms that have exponential worst case time-complexity. The complexity of the algorithm presented in [12, 13] is both exponential in time and space. The FPT algorithms whose time-complexity is at least exponential in the treewidth are other examples of this impracticability for large graphs: as far as we know, there exists no efficient implementation of the algorithm in [16] even for graphs with treewidth at most 4. On the positive side, efficient algorithms exist for computing the treewidth of particular graph classes, e.g., graphs with treewidth at most 4 [36]. Many heuristics for computing lower or upper bounds on the treewidth have also been designed [1, 18–20].

Surprisingly, much less work has been devoted to the computation of *pathwidth* and *path-decompositions* of graphs [6, 9, 28, 43, 49]. On the algorithmic point of view, it may sometimes be interesting to use path-decompositions of graphs rather than tree-decompositions. Indeed, many NP-hard problems are linear-time solvable in the class of graphs with bounded treewidth by dynamic programming on a tree-decomposition of the input graph [26]. In such algorithms, the most time-expensive steps are often the ones concerning the *branching* nodes (i.e., nodes with at least two children) in the decomposition. In case when the pathwidth of a graph is not much larger than its treewidth, performing dynamic programming on a path-decomposition would save time by avoiding such branching nodes. Another interest comes from the fact that the pathwidth of any n-node graph is at most $O(\log n)$ times its treewidth [9]. Hence,

*Extended abstract of this work has been presented at SEA 2014 [24]. This work has been partially supported by ANR project Stint under reference ANR-13-BS02-0007, ANR program "Investments for the Future" under reference ANR-11-LABX-0031-01, the associated Inria team AlDyNet and the project ECOS-Sud Chile.

providing an efficient algorithm to compute pathwidth immediately leads to an efficient approximation algorithm for computing the treewidth. Pathwidth has also been widely studied for its relationship with many applications such that rescuing games (search games [6,7,9,40,44]), space/time tradeoff for register allocations (pebble games [40]), VLSI design (gate matrix layout problem [27, 33]), etc.

During the last decade, several digraph decompositions have been proposed in order to try to bring to directed graphs the same algorithmic power as tree-decompositions provide in undirected graphs [4,37,38]. The directed counterpart of pathwidth has been defined by Reed, Thomas and Seymour (see [3]) and also has applications in search games [64, 65]. *Directed pathwidth* has also an interesting application in the context of routing reconfiguration in Wavelength-Division Multiplexing (WDM) networks. In this problem, the perturbation of the traffic induced by a re-allocation of the routes for some requests in a network can be measured by the directed pathwidth of the conflict digraph of these routes. Indeed, an optimal directed path-decomposition is equivalent to a reconfiguration minimizing the maximum number of simultaneous interruptions of the requests. Several heuristics as well as combinatorial and complexity results have been proposed on the directed pathwidth in this context [22, 23, 25, 56, 57].

In this paper, we design an algorithm that computes the directed pathwidth and a corresponding path-decomposition of directed graphs. We then provide an experimental analysis of our algorithm that presents a significant improvement with respect to existing algorithms.

1.1 Practical computation of (directed) pathwidth

A *path-decomposition* [3] of a directed graph $D = (V, A)$ is a sequence $P = (X_1, \cdots, X_r)$ of subsets of vertices of D such that:

1. $\bigcup_{1 \leq i \leq r} X_i = V$,
2. for any $1 \leq i \leq j \leq k \leq r$, $X_i \cap X_k \subseteq X_j$, and
3. for any arc (u, v), there exist $i \leq j$ such that $u \in X_i$ and $v \in X_j$.

The *width* of P is the size of the largest subset X_i minus one and the *directed pathwidth* of D, denoted by $dpw(D)$, is the minimum width among its path-decompositions. It is easy to see that, if D is *symmetric* (i.e., if $(u, v) \in A$ then $(v, u) \in A$ for all $u, v \in V$), then $dpw(D)$ equals the pathwidth of the underlying undirected graph. Hence, this definition generalizes the notion of pathwidth to directed graphs.

It is also easy to prove that any path-decomposition $P = (X_1, \cdots, X_r)$ can be turned into another path-decomposition $(Y_1, \cdots, Y_h)$ without increasing the width and such that $|Y_{i+1} \setminus Y_i| = 1$ for any $1 \leq i < h$. Therefore, any *path-decomposition* of a n-node digraph corresponds to a *layout* (i.e., an ordering) $(v_1, \cdots, v_n)$ of its vertices. Conversely, any layout of the vertices of a digraph defines a path-decomposition of it.

Computing the pathwidth of graphs is NP-hard in planar cubic graphs [47], in chordal graphs [35], and in bipartite distance hereditary graphs [42]. Therefore, computing the directed pathwidth is NP-hard in the class of symmetric digraphs. Computing the pathwidth is polynomial-time solvable in the class of cographs [21], permutation graphs [17], interval graphs, circular-arc graphs [59]. An exact algorithm for computing the pathwidth of n-node graphs in time $O(1.9657^n)$ is designed in [60], and an exact algorithm for computing the directed pathwidth of n-node digraphs in time $O(1.89^n)$ has been proposed in [41]. In the undirected case, an FPT algorithm has been proposed in [16] and some kernelization reduction rules are provided in [15]. On the practical side, to the best of our knowledge, very few implementations of algorithms for computing the pathwidth (exact or bounds) have been proposed.

Implementation of exact algorithms. Solano and Pióro proposed in [57] an exact Branch and Bound algorithm to compute the directed pathwidth of digraphs, that checks all possible layouts of the nodes and keeps a best one. A mixed integer linear programming formulation (MILP) has been proposed in [5, 57]. Another algorithm for computing the pathwidth is based on a SAT formulation of the problem that is solved using Constraint Programming solver [5]. None of these methods handle graphs with more than 30 nodes.

A dynamic programming algorithm (exponential both in time and space) for computing pathwidth of graphs is described in [12]. Two implementations of this algorithm have been proposed in [58, 61]. As far as we know, [61] is the single existing solution that manages to compute the pathwidth of some graphs with more than 30 nodes.

Heuristics. A polynomial-time heuristic for directed pathwidth is proposed in [23]. It aims at computing a layout of the nodes in a greedy way. At each step, the next node of the layout is chosen using a *flow circulation* method. A heuristic based on a Branch and Bound algorithm has been designed in [57]. Recently, a heuristic has been proposed that is based on the combination of a `Shake` function and `Local_Search` [30, 53]: it consists in sequentially improving a layout by switching the nodes until a local optimum is achieved.

1.2 Contributions and organization of the paper

We design a Branch and Bound algorithm that computes the exact directed pathwidth of digraphs and a corresponding path-decomposition. Basically, our algorithm explores the set of all possible layouts of the vertex-set, returning a layout with smallest width. Our main contribution consists of several non-trivial techniques to reduce the size of the input digraph (pre-processing) and to cut the exploration space during the search phase of the algorithm. Note that, some of the pre-processing rules are dedicated to symmetric digraphs (equivalently, they are dedicated for computing the pathwidth of undirected graphs). In Section 2, we prove several technical lemmas that allow us to prove the correctness of the pre-processing phase (Section 2.2) and of the pruning procedures (Section 2.3). We present our algorithm and prove its correctness in Section 3. Finally, in Section 4, we evaluate experimentally our algorithm by comparing it to existing algorithms of the literature. It appears from the simulations that our algorithm offers a significant gain with respect to previous work. It is able to compute the exact pathwidth of any graph with less than 60 nodes in a reasonable running-time ($\leq$ 10 minutes on a standard laptop). Moreover, our algorithm achieves good performance when used as a heuristic (i.e., when returning best result found within bounded time-limit).

2 Preliminaries

In this section, we formally define the notion of directed pathwidth in terms of directed *vertex-separation*. Then, we give some technical lemmas that are used to prove the correctness of our algorithm.

2.1 Definitions and Notations

All graphs and digraphs considered in this paper are connected and loopless. Let $G = (V, E)$ be a graph. For any set $S \subseteq V$, let $N_G(S)$ be the set of nodes in $V \setminus S$ that have a neighbor in S. For any arc $(u, v) \in A$ of a digraph $D = (V, A)$, u is called an *in-neighbor of* v and v is an *out-neighbor* of u. For any digraph $D = (V, A)$ and any subset $S \subseteq V$, let $N^+_D(S)$ (resp., $N^-_D(S)$) be the set of nodes in $V \setminus S$ that have an in-neighbor (resp., an out-neighbor) in S. In other words, $N^+_D(S)$ (resp., $N^-_D(S)$) is the set of nodes in $V \setminus S$ that are out-neighbors (resp., in-neighbors) of some vertex in S. We omit the subscript when there is no ambiguity. For any digraph $D = (V, A)$ and $a = uv \in A$, let D/a be the digraph obtained from D by contracting a. Let $x_a = x_{uv}$ denote the vertex of D/a that results from the identification of u and v. Let $D \setminus u$ be the digraph obtained from D by removing u and its incident arcs.

A *strongly connected component* C of a digraph D is a inclusion-maximal subgraph of D such that, for any two vertices $u, v \in V(C)$, there is a directed path from u to v in C. Abusing the notations, we make confusion between C and $V(C)$.

Layouts and vertex-separation. As observed in the Introduction, path-decompositions of (di)graphs can be defined in terms of vertices layouts. More precisely, the pathwidth of a graph is known to be equal to its *vertex separation* [39]. This result can easily be extended to directed graphs, i.e., the directed pathwidth equals its directed vertex separation [64]. In what follows, we only use the layout terminology as defined below.

Given a set S, a *layout* of S is any ordering of the elements of S. Let $\mathcal{L}(S)$ denote the set of all layouts of S. Let $S' \subseteq S$ and P, Q be two layouts of S' and $S \setminus S'$ respectively. Let $P \odot Q$ be the layout of S obtained by concatenating P and Q. Moreover, let $\mathcal{L}_P(S) = \{L \in \mathcal{L}(S) \mid L = P \odot Q, Q \in \mathcal{L}(S \setminus S')\}$, i.e., $\mathcal{L}_P(S)$ is the set of all layouts of S with prefix P.

Let $D = (V, A)$ be a digraph and let $L = (v_1, \cdots, v_{|S|}) \in \mathcal{L}(S)$ be a layout of $S \subseteq V$. For any $1 \leq i \leq |S|$, let $\nu(L, i) = |N^+_D(\{v_1, \cdots, v_i\})|$ and $\nu(L) = \max_{i \leq |S|} \nu(L, i)$. The *directed vertex-separation* of D,

denoted by $vs(D)$, is equal to the minimum $\nu(L)$ among all layouts L of V, i.e., $vs(D) = \min_{L \in \mathcal{L}(V)} \nu(L)$[1]. In what follows, we will omit "directed" and simply call this parameter "vertex separation". For any digraph D, $vs(D) = dpw(D)$ [64].

2.2 Technical lemmas for preprocessing

In this section, we prove some technical lemmas that will be useful to prove the correctness of the preprocessing part of our algorithm.

Next lemma shows that we can restrict our study to strongly connected digraphs. We provide its proof for completeness.

Lemma 1 (Folklore) *Let D be any connected digraph and let $SCC(D)$ be the set of strongly connected components of D. Then, $vs(D) = \max_{D' \in SCC(D)} vs(D')$.*

Proof. Let L be any layout of D and let $D' \in SCC(D)$. Let L' be the corresponding layout of D' (that is, for any $u, v \in V(D')$, u appears before v in L' if and only if u appears before v in L). Then, $\nu(L') \leq \nu(L)$.

Conversely, for any $D' \in SCC(D)$, let $L_{D'}$ be an optimal layout of D'. For any $D', D'' \in SCC(D)$, we set $D' \prec D''$ if there is a path from D'' to D'. Note that, in this case, there is no path from D' to D'', and therefore $\prec$ is a partial order. Let L be the layout of D obtained by concatenating the layouts in $(L_{D'})_{D' \in SCC(D)}$ in such a way that $L_{D''}$ is after $L_{D'}$ if $D' \prec D''$ for all $D', D'' \in SCC(D)$ (this is possible since $\prec$ is a partial order). Then $\nu(L) \leq \max_{D' \in SCC(D)} \nu(L_{D'})$. □

In what follows, we show that we can contract some well chosen arcs of a digraph without modifying its vertex-separation. It is well known that, in undirected graphs, edge-contraction cannot increase the pathwidth. However, this is not true anymore in digraphs as shown in the following example. Let $D = (\{a, b, c, d\}, \{ab, cb, cd, da\})$. D is acyclic and therefore, $vs(D) = 0$. On the other hand, D/cb is a directed cycle with 3 nodes and $vs(D/cb) = 1$.

Lemmas 2 and 3 provide explicit conditions under which arc-contractions can be done without increasing the vertex-separation.

Lemma 2 *Let $D = (V, A)$ be a n-node digraph and $uv \in A$ such that either $vu \in A$ or $N^-(v) = \{u\}$. Then, $vs(D) \geq vs(D/uv)$.*

Proof. Let $L = (v_1, \cdots, v_n)$ be a layout of V with $\{u, v\} = \{v_i, v_j\}$ and $i < j$.

Let us consider the layout $L' = (v'_1, \cdots, v'_{n-1}) = (v_1, \cdots, v_{i-1}, v_{i+1}, \cdots, v_{j-1}, x_{uv}, v_{j+1}, \cdots, v_n)$ of $V(D/uv)$[2]. Let $1 \leq k < n-1$, $w \in N^+_{D/uv}(v'_1, \cdots, v'_k)$ and let $x \in \{v'_1, \cdots, v'_k\}$ such that $xw \in A(D/uv)$.

Let $N^k = N^+_D(v_1, \cdots, v_k)$ if $k < i$, and $N^k = N^+_D(v_1, \cdots, v_{k+1})$ if $i \leq k < j-1$, and $N^k = N^+_D(v_1, \cdots, v_{k+2})$ if $k \geq j-1$.

- If $w \neq x_{uv}$, then
  - either $x \neq x_{uv}$ and $xw \in A(D)$ and $w \in N^k$.
  - or $x = x_{uv}$, then $k \geq j-1$, $w \notin \{u, v\}$ and either $uw \in A(D)$ or $vw \in A(D)$. Therefore, $w \in N^k$.
- If $w = x_{uv}$ then $k < j-1$ and either $xu \in A(D)$ or $xv \in A(D)$.
  - if $k < i$, then either u or v belong to N^k since $xu \in A(D)$ or $xv \in A(D)$.
  - otherwise, $i \leq k < j-1$ and, by hypothesis on u and v:
    - either $v_i v_j \in A$, in which case $v_j \in N^k$,
    - or $v_i = v$, $v_j = u$ and we have that $xu \in A(D)$, since $N^-(v) = \{u\}$ and $x \neq u$ (i.e., $xv \notin A(D)$). Therefore, $xu \in A(D)$ and $u \in N^k$.

[1] Note that a different slightly definition of directed vertex separation of DAGs has been introduced in [14]

[2] Here, to simplify the presentation, we slightly abuse the notations by identifying the nodes of $D \setminus \{u, v\}$ and the nodes of $(D/uv) \setminus \{x_{uv}\}$.

In all cases, we get that $|N^+_{D/uv}(v'_1, \cdots, v'_k)| \leq |N^k|$ and therefore, for all layouts L of D there is a layout L' of D/uv such that $\nu(L') \leq \nu(L)$. Hence, $vs(D/uv) \leq vs(D)$. □

Lemma 3 *Let $D = (V, A)$ be a n-node digraph and $u \in V$ such that $N^+(u) = \{v\}$. Then, $vs(D) \geq vs(D/uv)$.*

Proof. Let $L = (v_1, \cdots, v_n)$ be a layout of V with $\{u, v\} = \{v_i, v_j\}$ and $i < j$. The case $v_iv_j \in A$ is similar to the one of previous lemma. Therefore, let us assume that $v_i = v$ and $vu \notin A$. Let us consider the layout $L' = (v'_1, \cdots, v'_{n-1}) = (v_1, \cdots, v_{i-1}, x_{uv}, v_{i+1}, \cdots, v_{j-1}, v_{j+1}, \cdots, v_n)$ of $V(D/uv)$. Let $1 \leq k < n-1$, $w \in N^+_{D/uv}(v'_1, \cdots, v'_k)$ and let $x \in \{v'_1, \cdots, v'_k\}$ such that $xw \in A(D/uv)$.

Let $N^k = N^+_D(v_1, \cdots, v_k)$ if $k < i$, $N^k = N^+_D(v_1, \cdots, v_{i-1}, v = v_i, \cdots, v_k)$ if $i \leq k < j$, and $N^k = N^+_D(v_1, \cdots, v_{i-1}, v, v_{i+1}, \cdots, v_{j-1}, u, v_{j+1}, \cdots, v_{k+1})$ if $k \geq j$.

- If $w, x \neq x_{uv}$, then $xw \in A(D)$ and $w \in N^k$.
- If $w = x_{uv}$ then $k < i$, $x \notin \{u, v\}$, and either $xu \in A(D)$ or $xv \in A(D)$. Hence, u or v belongs to N^k.
- If $x = x_{uv}$ then $k \geq i$, $w \notin \{u, v\}$, either $uw \in A(D)$ or $vw \in A(D)$. But v being the single out-neighbor of u and $v \neq w$, we get that $vw \in A(D)$, and therefore, $w \in N^k$.

In all cases, we get that $|N^+_{D/uv}(v'_1, \cdots, v'_k)| \leq |N^k|$ and therefore, for all layouts L of D there is a layout L' of D/uv such that $\nu(L') \leq \nu(L)$. Hence, $vs(D/uv) \leq vs(D)$. □

Next lemma show that under some conditions, the vertex-separation does not decrease after arc-contraction.

Lemma 4 *Let $D = (V, A)$ be a n-node digraph and $uv \in A$ such that,*

- *$vu \notin A$* *(no loops are created during the contraction)*
- *and*
  - *either $N^-(v) = \{u\}$ and $N^+(u) \cap N^+(v) = \emptyset$, (v has in-degree 1 and no parallel arcs out-going from x_{uv} are created during the contraction)*
  - *or $N^+(u) = \{v\}$ and $N^-(u) \cap N^-(v) = \emptyset$ (u has out-degree 1 and no parallel arcs in-going to x_{uv} are created during the contraction).*

Then, $vs(D) \leq vs(D/uv)$.

Proof. Let $L' = (v_1, \cdots, v_{i-1}, x_{uv}, v_{i+1}, \cdots, v_{n-1})$ be any optimal layout of D/uv.

Case: $v_kv \notin A$ for any $k < i$, i.e., $v \notin N^+_D(v_1, \cdots, v_{i-1})$. Note that, in particular, it is the case when $N^-(v) = \{u\}$.

Let $L_1 = (v_1, \cdots, v_{i-1}, v, u, v_{i+1}, \cdots, v_{n-1})$ and $L_2 = (v_1, \cdots, v_{i-1}, u, v, v_{i+1}, \cdots, v_{n-1})$.

We show that $\nu(L_1) \leq \nu(L')$ or $\nu(L_2) \leq \nu(L')$.

1. Let $j < i$ and $w \in N^+_D(v_1, \cdots, v_j)$. Since $v \notin N^+_D(v_1, \cdots, v_j)$, $w \neq v$.
   - If $w \neq u$, then $w \in N^+_{D/uv}(v_1, \cdots, v_j)$.
   - Otherwise, there is v_h, $h \leq j$ such that $v_hu \in A(D)$ and so $v_hx_{uv} \in A(D/uv)$ and, thus, $x_{uv} \in N^+_{D/uv}(v_1, \cdots, v_j)$.

   Hence, in both cases, $|N^+_D(v_1, \cdots, v_j)| \leq |N^+_{D/uv}(v_1, \cdots, v_j)|$.
2. Let $j > i$ and $w \in N^+_D(v_1, \cdots, v_j)$. Since $w \notin \{u, v\}$, then $w \in N^+_{D/uv}(v_1, \cdots, v_{i-1}, x_{uv}, v_{i+1}, \cdots, v_j)$ and, again, $|N^+_D(v_1, \cdots, v_j)| \leq |N^+_{D/uv}(v_1, \cdots, v_j)|$.

3. By definition of x_{uv},

$$|N_D^+(v_1,\cdots,v_{i-1},u,v)| = |N_D^+(v_1,\cdots,v_{i-1},v,u)| = |N_{D/uv}^+(v_1,\cdots,v_{i-1},x_{uv})|.$$

4. It remains to show that $|N_D^+(v_1,\cdots,v_{i-1},v)| \leq \nu(L')$ or $|N_D^+(v_1,\cdots,v_{i-1},u)| \leq \nu(L')$
First, note that $N_D^+(v_1,\cdots,v_{i-1},v) = (N_D^+(v_1,\cdots,v_{i-1})\setminus\{v\})\cup(N_D^+(v)\cap\{u,v_{i+1},\cdots,v_{n-1}\})$. Moreover, since $v \notin N_D^+(v_1,\cdots,v_{i-1})$ and $vu \notin A$, this implies that

$$N_D^+(v_1,\cdots,v_{i-1},v) = N_D^+(v_1,\cdots,v_{i-1}) \cup (N_D^+(v)\cap\{v_{i+1},\cdots,v_{n-1}\}) \tag{1}$$

On the other hand, by definition, we have that

$$N_D^+(v_1,\cdots,v_{i-1},v,u) = (N_D^+(v_1,\cdots,v_{i-1})\setminus\{u\})\bigcup((N_D^+(v)\cup N_D^+(u))\cap\{v_{i+1},\cdots,v_{n-1}\}) \tag{2}$$

- If $N_D^+(v)\cap\{v_{i+1},\cdots,v_{n-1}\} \subseteq N_D^+(v_1,\cdots,v_{i-1})$, then by Equation (1) we have $N_D^+(v_1,\cdots,v_{i-1},v) = N_D^+(v_1,\cdots,v_{i-1})$. Hence, by item (1), we get that $|N_D^+(v_1,\cdots,v_{i-1},v)| = |N_D^+(v_1,\cdots,v_{i-1})| \leq |N_{D/uv}^+(v_1,\cdots,v_{i-1})|$.
- Else, let us assume that $(N_D^+(v)\cap\{v_{i+1},\cdots,v_{n-1}\})\setminus N_D^+(v_1,\cdots,v_{i-1}) \neq \emptyset$.
  - If, either $(N_D^+(u)\setminus N_D^+(v_1,\cdots,v_{i-1},v))\cap\{v_{i+1},\cdots,v_{n-1}\} \neq \emptyset$ or $u \notin N_D^+(v_1,\cdots,v_{i-1})$, then, by Equations (1) and (2), we get $|N_D^+(v_1,\cdots,v_{i-1},v)| \leq |N_D^+(v_1,\cdots,v_{i-1},v,u)| \leq |N_{D/uv}^+(v_1,\cdots,x_{uv})|$ (where the last inequality comes by item (3))
  - Otherwise, since either $N^+(u)\cap N^+(v) = \emptyset$ or $N^+(u) = \{v\}$, we get that $N^+(u)\cap\{v,v_{i+1},\cdots,v_{n-1}\} \subseteq N_D^+(v_1,\cdots,v_{i-1})\cup\{v\}$.
    Moreover, $u \in N_D^+(v_1,\cdots,v_{i-1})$. Hence, $N_D^+(v_1,\cdots,v_{i-1},u) = N_D^+(v_1,\cdots,v_{i-1})\cup\{v\}\setminus\{u\}$ and $|N_D^+(v_1,\cdots,v_{i-1},u)| = |N_D^+(v_1,\cdots,v_{i-1})|$. By item (1), we get that $|N_D^+(v_1,\cdots,v_{i-1},u)| \leq |N_{D/uv}^+(v_1,\cdots,v_{i-1})|$.

Hence, $vs(D) \leq \min\{\nu(L_1),\nu(L_2)\} \leq \nu(L') = vs(D/uv)$.

Case: there is $k<i$ such that $v_kv \in A$. In that case, we have $N^+(u) = \{v\}$ and $N^-(u)\cap N^-(v) = \emptyset$.
Let ℓ be the smallest integer such that $v_\ell v \in A$ or $v_\ell u \in A$.

Note that, since $N^-(u)\cap N^-(v) = \emptyset$, either ($\ell = k$ and $v_\ell v \in A$ and $v_\ell u \notin A$), or ($\ell < k$ and $v_\ell u \in A$ and $v_\ell v \notin A$).

Let $L = (v_1,\cdots,v_\ell,u,v_{\ell+1},\cdots,v_{i-1},v,v_{i+1},\cdots,v_{n-1})$.

We show that $\nu(L) \leq \nu(L')$.

1. Let $j<\ell$ and $w \in N_D^+(v_1,\cdots,v_j)$. Since $v,u \notin N_D^+(v_1,\cdots,v_j)$, $w \notin \{u,v\}$. Therefore, $w \in N_{D/uv}^+(v_1,\cdots,v_j)$. Hence, $|N_D^+(v_1,\cdots,v_j)| \leq |N_{D/uv}^+(v_1,\cdots,v_j)|$.
2. By definition of ℓ, either u or v (not both) is in $N_D^+(v_1,\cdots,v_\ell)$. Also, $x_{uv} \in N_{D/uv}^+(v_1,\cdots,v_\ell)$. Moreover, for any $w \in N_D^+(v_1,\cdots,v_\ell)\setminus\{u,v\}$, we have that $w \in N_{D/uv}^+(v_1,\cdots,v_\ell)$. Hence, $|N_D^+(v_1,\cdots,v_\ell)| \leq |N_{D/uv}^+(v_1,\cdots,v_\ell)|$.
3. Let $j>i$ and $w \in N_D^+(v_1,\cdots,v_\ell,u,v_{\ell+1},\cdots,v_{i-1},v,v_{i+1},\cdots,v_j) = P$, then $w \in N_{D/uv}^+(v_1,\cdots,v_{i-1},x_{uv},v_{i+1},\cdots,v_j)$ and, again, $|P| \leq |N_{D/uv}^+(v_1,\cdots,v_j)|$.
4. Let $\ell<j<i$. Note that, $v \in N_D^+(v_1,\cdots,v_\ell,u,\cdots,v_j) = P$ and $x_{uv} \in N_{D/uv}^+(v_1,\cdots,v_j)$.
   Moreover, for any $w \in P\setminus\{v\}$, we have $w \in N_{D/uv}^+(v_1,\cdots,v_j)$ because $N^+(u) = \{v\}$.
   Hence, $|P| \leq |N_{D/uv}^+(v_1,\cdots,v_j)|$.
5. Moreover, by definition of x_{uv}, $|N_D^+(v_1,\cdots,v_\ell,u,v_{\ell+1},\cdots,v_{i-1},v)| = |N_{D/uv}^+(v_1,\cdots,x_{uv})|$.
6. Finally, either $v_\ell u \in A$, and then $N_D^+(v_1,\cdots,v_\ell,u) = N_D^+(v_1,\cdots,v_\ell)\cup\{v\}\setminus\{u\}$, or $v_\ell v \in A$, and then $N_D^+(v_1,\cdots,v_\ell,u) = N_D^+(v_1,\cdots,v_\ell)$.
   In both cases, $|N_D^+(v_1,\cdots,v_\ell,u)| \leq |N_D^+(v_1,\cdots,v_\ell)| \leq |N_{D/uv}^+(v_1,\cdots,v_\ell)|$ (the second inequality comes from the second item).

Hence, $vs(D) \leq \nu(L) \leq \nu(L') = vs(D/uv)$.

□

Altogether, we get from previous lemmas:

Theorem 1 *Let $D = (V, A)$ be a n-node digraph and $uv \in A$ such that, $vu \notin A$, and either ($N^-(v) = \{u\}$ and $N^+(u) \cap N^+(v) = \emptyset$) or ($N^+(u) = \{v\}$ and $N^-(u) \cap N^-(v) = \emptyset$). Then, $vs(D) = vs(D/uv)$. Moreover, an optimal layout for D can be obtained in linear time from an optimal layout of D/uv.*

The next lemma is almost trivial in undirected graphs. We extend it to digraphs. Loosely speaking, the next lemma say that, if a digraph D contains a symmetric path (a, b, c) as induced sub-digraph, any link incident to a, b or c is symmetric, and a, b and c have degree at most 2 in the underlying undirected graph (in particular, b has degree exactly 2), then contracting the arc (b, c) does not decrease the vertex separation. This situation is depicted in Figure 1.

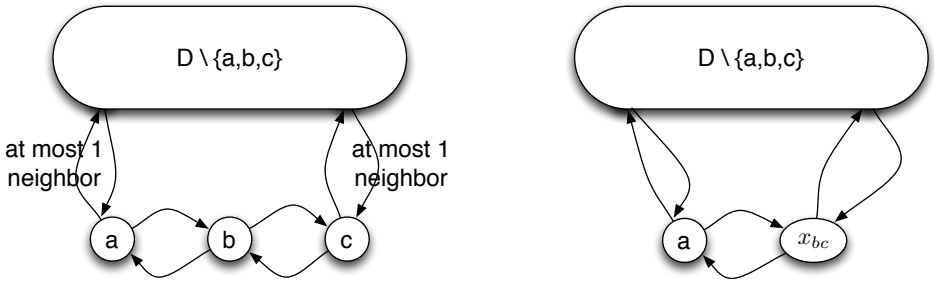


Figure 1: Description of the situation in Lemma 5: before contraction (left) and after contraction (right).

Lemma 5 *Let $D = (V, A)$ be a n-node digraph and let $a, b, c \in V$ be three nodes with $N^+_D(b) = N^-_D(b) = \{a, c\}$, $N^+_D(a) = N^-_D(a)$ and $N^+_D(c) = N^-_D(c)$. Moreover, $|N^+_D(a)| \leq 2$, $|N^+_D(c)| \leq 2$ and $c \notin N^+_D(a)$.*
Then, $vs(D) = vs(D/bc)$.

Proof. If a and c have only b as a neighbor, the result holds trivially. Otherwise, w.l.o.g., let us assume that $y \in N^+(c) \setminus \{b\}$ exists. Since $cb \in A$, by Lemma 2, $vs(D/bc) \leq vs(D)$.

Let $L = (v_1, \cdots, v_{n-1})$ be a layout of D/bc. We first show that there is a layout L^* of D/bc such that $\nu(L^*) \leq \nu(L)$ and, in L^*, x_{bc} appears between a and y. Then, we will show that there is a layout L' of D such that $\nu(L') \leq \nu(L^*)$.

Let $v_i = a$, $v_j = x_{bc}$ and $v_k = y$. If x_{bc} is between a and y in L, i.e., either $i < j < k$ or $k < j < i$, then let us set $L^* = L$. Otherwise, let us assume that $k < i < j$ (the other cases, when $i < k < j$ or $j < i < k$ or $j < k < i$, can be dealt with similarly by symmetry).

Then, consider the layout $L^* = (v_1, \cdots, v_{i-1}, v_j = x_{bc}, v_i = a, \cdots, v_{j-1}, v_{j+1}, \cdots, v_{n-1})$ of D/bc. We show that $\nu(L^*) \leq \nu(L)$. Indeed, $N^+_{D/bc}(v_1, \cdots, v_{i-1}, x_{bc}) \subseteq N^+_{D/bc}(v_1, \cdots, v_{i-1}) \cup \{a\} \setminus \{x_{bc}\}$ (because $k < i$) and, since $x_{bc} \in N^+_{D/bc}(v_1, \cdots, v_{i-1})$, we get $|N^+_{D/bc}(v_1, \cdots, v_{i-1}, x_{bc})| \leq |N^+_{D/bc}(v_1, \cdots, v_{i-1})|$. Moreover, since x_{bc} has no out-neighbor in $\{v_{i+1}, \cdots, v_{n-1}\}$, we have
$N^+_{D/bc}(v_1, \cdots, v_{i-1}, x_{bc}, v_i, \cdots, v_h) \subseteq N^+_{D/bc}(v_1, \cdots, v_h) \setminus \{x_{bc}\}$ for any $i \leq h < n$.

Hence, we may assume that $L^* = (v_1, \cdots, v_{n-1})$ is a layout of D/bc where $i < j < k$ (the case $k < j < i$ is symmetric).

Let us consider the layout $L' = (v_1, \cdots, v_i = a, b, v_{i+1}, \cdots, v_{j-1}, c, v_{j+1}, \cdots, v_{n-1})$ of V.

- For all $1 \leq h < i$, $|N^+_D(v_1, \ldots, v_h)| = |N^+_{D/bc}(v_1, \ldots, v_h)|$ since $b \notin N^+_D(v_1, \ldots, v_h)$.
- $|N^+_D(v_1, \ldots, v_i)| = |N^+_{D/bc}(v_1, \ldots, v_i)|$ since $x_{bc} \in N^+_{D/bc}(v_1, \ldots, v_i)$, $b \in N^+_D(v_1, \ldots, v_i)$ and, because $y = v_k$ with $k > i$, $c \notin N^+_D(v_1, \ldots, v_i)$.

- Similarily, $|N^+_D(v_1,\ldots,v_i,b)| = |N^+_{D/bc}(v_1,\ldots,v_i)|$.
- Finally, for any $i < h < n$, $|N^+_D(v_1,\ldots,v_h)| = |N^+_{D/bc}(v_1,\ldots,v_h)|$.

Hence, $\nu(L') \leq \nu(L^*)$. □

2.3 Technical lemmas for the *Pruning* part

In this section, we prove some technical lemmas that will be useful to prove the correctness of the *Pruning* part of our *Branch & Bound* algorithm.

When looking for a good layout of the nodes of a digraph, our algorithm will, at some step, consider a layout P of some subset $S \subset V$ and look for the best layout L of V starting with P. Next lemma gives some conditions on a node $v \in V \setminus S$ to ensure that the best solution starting with $P \odot v$ is as good as L.

Lemma 6 *Let $D = (V, A)$ be a n-node digraph, $S \subset V$, and $P \in \mathcal{L}(S)$. If there exists $v \in V \setminus S$ such that either $N^+(v) \subseteq (S \cup N^+(S))$, or $v \in N^+(S)$ and there exists a vertex w such that $N^+(v) \setminus (S \cup N^+(S)) = \{w\}$. Then, $\min_{L \in \mathcal{L}_P(V)} \nu(L) = \min_{L \in \mathcal{L}_{P \odot \{v\}}(V)} \nu(L)$.*

Proof. Note that, by definition, $\min_{L \in \mathcal{L}_P(V)} \nu(L) \leq \min_{L \in \mathcal{L}_{P \odot v}(V)} \nu(L)$.

Let $P = (v_1, \cdots, v_{|S|})$ and let $Q = (v_{|S|+1}, \cdots, v_n) \in \mathcal{L}(V \setminus S)$ such that $\nu(P \odot Q) = \min_{L \in \mathcal{L}_P(V)} \nu(L)$. Finally, let $|S| < r \leq n$ such that $v = v_r$.

If $N^+(v) \subseteq S \cup N^+(S)$, then $N^+(v_1, \cdots, v_{|S|}, v) \subseteq N^+(v_1, \cdots, v_{|S|})$ and, for any $|S| < j \leq r$, $N^+(v_1, \cdots, v_{|S|}, v, v_{|S|+1}, \cdots, v_j) \subseteq N^+(v_1, \cdots, v_j)$.

If $v \in N^+(S)$ and there exists a vertex w such that $N^+(v) \setminus (S \cup N^+(S)) = \{w\}$, then $N^+(v_1, \cdots, v_{|S|}, v) = N^+(v_1, \cdots, v_{|S|}) \cup \{w\} \setminus \{v\}$ and, for any $|S| < j < r$, $N^+(v_1, \cdots, v_{|S|}, v, v_{|S|+1}, \cdots, v_j) \subseteq N^+(v_1, \cdots, v_j) \cup \{w\} \setminus \{v\}$.

In both cases, for any j, $r \leq j \leq n$, then $N^+(v_1, \cdots, v_{|S|}, v, v_{|S|+1}, \cdots, v_j) = N^+(v_1, \cdots, v_j)$.

Hence,

$$\begin{aligned}
\min_{L \in \mathcal{L}_{P \odot v}(V)} \nu(L) &\leq \nu(P \odot v \odot (v_{|S|+1}, \cdots, v_{r-1}, v_{r+1}, \cdots, v_n)) \\
&= \max\big\{\nu(P);\ |N^+(v_1, \cdots, v_{|S|}, v)|;\ \max_{|S|<j<r} |N^+(v_1, \cdots, v_{|S|}, v, v_{|S|+1}, \cdots, v_j)|; \\
&\qquad\qquad \max_{r<j\leq n} |(N^+(v_1, \cdots, v_j)|\big\} \\
&\leq \max\big\{\nu(P);\ |N^+(v_1, \cdots, v_{|S|})|;\ \max_{|S|<j\leq r} |N^+(v_1, \cdots, v_j)|;\ \max_{r<j\leq n} |N^+(v_1, \cdots, v_j)|\big\} \\
&= \nu(P \odot Q) \\
&= \min_{L \in \mathcal{L}_P(V)} \nu(L)
\end{aligned}$$

□

Our *Branch & Bound* algorithm presented in next section considers *a priori* all possible layouts of the vertices of the input digraph. To speed it up, we aim at removing some of the layouts from the search space. For this purpose, we record some partial orderings that have led to a solution whose cost is larger than the one of the best solution obtained so far. During the exploration of an ordering, the algorithm compares its prefix to the partial orderings already stored: if the prefix is a permutation of such a partial ordering, next lemma proves that it is useless to pursue the exploration of this branch. Somehow, the idea behind Lemma 7 is similar to the technique used by the dynamic programming algorithm proposed in [12] in order to reduce the number of subsets to be considered.

Let L be the best layout of V having a prefix P. We express some conditions under which no layout of V starting with a permutation of P is better than L.

Lemma 7 *Let $D = (V, A)$ be a n-node digraph, $S \subset V$ and let $P, P' \in \mathcal{L}(S)$ be two layouts of S. If $\nu(P) < \min_{L \in \mathcal{L}_P(V)} \nu(L)$ or $\nu(P) \leq \nu(P')$, then $\min_{L \in \mathcal{L}_P(V)} \nu(L) \leq \min_{L \in \mathcal{L}_{P'}(V)} \nu(L)$.*

Proof. Let $r = |S|$ and let $P = (v_1, \cdots, v_r), P' = (v'_1, \cdots, v'_r) \in \mathcal{L}(S)$. Note that $V(P) = V(P') = S$. Let $Q = (v_{r+1}, \cdots, v_n) \in \mathcal{L}(V \setminus S)$ such that $\nu(P' \odot Q) = \min_{L \in \mathcal{L}_{P'}(V)} \nu(L)$.

Note that, by definition, $\nu(P) \leq \min_{L \in \mathcal{L}_P(V)} \nu(L)$.

- Let us assume first that $\nu(P) < \min_{L \in \mathcal{L}_P(V)} \nu(L) \leq \nu(P \odot Q)$.

$$\begin{aligned}
\min_{L \in \mathcal{L}_P(V)} \nu(L) &\leq \nu(P \odot Q) \\
&= \max_{1 \leq i \leq n} |N^+(v_1, \cdots, v_i)| \\
&= \max\{\max_{1 \leq i \leq r} |N^+(v_1, \cdots, v_i)|; \max_{r < i \leq n} |N^+(v_1, \cdots, v_i)|\} \\
&= \max\{\nu(P); \max_{r < i \leq n} |N^+(v_1, \cdots, v_r, v_{r+1}, \cdots, v_i)|\} \\
&= \max_{r < i \leq n} |N^+(v_1, \cdots, v_i)| && (\textit{because } \nu(P) < \nu(P \odot Q)) \\
&= \max_{r < i \leq n} |N^+(v'_1, \cdots, v'_r, v_{r+1}, \cdots, v_i)| && (\textit{because } V(P) = V(P')) \\
&\leq \max\{\nu(P'); \max_{r < i \leq n} |N^+(v'_1, \cdots, v'_r, v_{r+1}, \cdots, v_i)|\} \\
&= \nu(P' \odot Q) \\
&= \min_{L \in \mathcal{L}_{P'}(V)} \nu(L)
\end{aligned}$$

- Otherwise, $\nu(P) = \min_{L \in \mathcal{L}_P(V)} \nu(L)$ and $\nu(P) \leq \nu(P')$.

$$\begin{aligned}
\min_{L \in \mathcal{L}_P(V)} \nu(L) &= \nu(P) \\
&\leq \max\{\nu(P); \max_{r < i \leq n} |N^+(v_1, \cdots, v_r, v_{r+1}, \cdots, v_i)|\} \\
&= \max\{\nu(P); \max_{r < i \leq n} |N^+(v'_1, \cdots, v'_r, v_{r+1}, \cdots, v_i)|\} \ (\textit{because } V(P) = V(P')) \\
&\leq \max\{\nu(P'); \max_{r < i \leq n} |N^+(v'_1, \cdots, v'_r, v_{r+1}, \cdots, v_i)|\} \ (\textit{because } \nu(P) \leq \nu(P')) \\
&= \nu(P' \odot Q) \\
&= \min_{L \in \mathcal{L}_{P'}(V)} \nu(L)
\end{aligned}$$

□

3 The algorithm

In this section, we present our exact exponential-time algorithm for computing $vs(D)$ for any n-node digraph $D = (V, A)$. To ease the presentation, we assume that D is strongly connected. If this is not the case, we apply all the steps of our algorithm to each of its strongly connected components, and then use the construction presented in the proof of Lemma 1 to deduce in linear time the layout L of D such that $\nu(L) = vs(D)$.

The algorithm starts by a pre-processing phase in order to reduce the size of the input. Then, our algorithm is based on a *Branch* & *Bound* procedure.

3.1 Pre-processing and Post-processing phases

In this section, we describe some simple rules that, for any (di)graph D, allows to compute a smaller (di)graph D^* (pre-processing) such that an optimal layout for D can easily be obtained from an optimal layout of D^* (post-processing). We give the pre-processing rules to build D^* from D as well as the way to build an optimal layout of D from an optimal layout of D^* in parallel.

The following reduction rules are applied while it is possible. The rules that have been proposed for undirected graphs can be used only when D is a symmetric digraph. Note that, the digraph D^*

obtained after applying once one of these rules is strongly connected, and that given the layout L^* such that $\nu(L^*) = vs(D^*)$, one can deduce in linear time a layout L for D such that $\nu(L) = vs(D)$.

In what follows, we formally define the reduction rules and explain how to get an optimal layout L for D from an optimal layout L^* of the reduced graph D^*. Given any layout $L = (v_1, \cdots, v_n)$, let us set $L^{-1}(v_i) = i$ for any $1 \leq i \leq n$.

Rule A: If there is $uv \in A$ such that, $vu \notin A$, and either ($N^-(v) = \{u\}$ and $N^+(u) \cap N^+(v) = \emptyset$), or ($N^+(u) = \{v\}$ and $N^-(u) \cap N^-(v) = \emptyset$), then let $D^* = D/uv$ (By Theorem 1, $vs(D^*) = vs(D)$).

Let $L^* = (v_1, \cdots, v_{i-1}, x_{uv}, v_{i+1}, \cdots, v_{n-1}) = P \odot x_{uv} \odot Q$ be an optimal layout of D^* (i.e., $\nu(L^*) = vs(D^*)$). Then, a layout L of D such that $\nu(L) = vs(D)$ can be computed from L^* as follows (see the proof of Theorem 1).

- If $v \notin N_D^+(P)$, then
  - if $N_D^+(v) \cap Q \subseteq N_D^+(P)$ or $(N_D^+(u) \setminus N_D^+(P \cup \{v\})) \cap Q \neq \emptyset$ or $u \notin N_D^+(P)$, then let $L = P \odot v \odot u \odot Q$.
  - else, let $L = P \odot u \odot v \odot Q$.
- Else, let ℓ be the smallest integer such that $v_\ell v \in A$ or $v_\ell u \in A$, and now, let $L^* = P_1 \odot v_\ell \odot P_2 \odot x_{uv} \odot Q$. We obtain $L = P_1 \odot v_\ell \odot u \odot P_2 \odot v \odot Q$.

Rule B: If there are three nodes $a, b, c \in V$ with $N_D^+(b) = N_D^-(b) = \{a, c\}$, $N_D^+(a) = N_D^-(a) = \{b, x\}$, $N_D^+(c) = N_D^-(c) = \{b, y\}$, and $x \neq c$, then let $D^* = D/bc$ (By Lemma 5, $vs(D^*) = vs(D)$).

Let $L^* = (v_1, \cdots, v_{i-1}, x_{bc}, v_{i+1}, \cdots, v_{n-1})$ be an optimal layout of D^* (i.e., $\nu(L^*) = vs(D^*)$). Then, as shown in the proof of Lemma 5, a layout L of D such that $\nu(L) = vs(D)$ can be computed from L^* as follows.

First, if x_{bc} appears between a and y in L^*, let $L' = L^*$. Otherwise, L' is obtained from L^* by removing x_{bc} and re-inserting it just after the first of y and a. For instance, if $L^* = P \odot a \odot Q \odot y \odot R \odot x_{bc} \odot S$, then $L' = P \odot a \odot x_{bc} \odot Q \odot y \odot R \odot S$.

Finally, L is obtained from L' by replacing x_{bc} by b then c if y is after x_{bc} in L', or by replacing x_{bc} by c then b if y is before x_{bc} in L'. For instance, if $L^* = P \odot x_{bc} \odot Q \odot y \odot R$ then we obtain $L = P \odot b \odot c \odot Q \odot y \odot R$.

Rule C: If D is symmetric, let G be the underlying undirected graph. It is shown in [15] that $vs(D^*) = vs(D)$ when D^* is obtained as follows.

C.1: If (in G) two degree-one vertices u and v share the same neighbor w, then $D^* = D \setminus u$.

To compute an optimal layout L for D from an optimal layout $L^* = P \odot w \odot Q$ for D^*, it is sufficient to insert u before w, i.e., $L = P \odot u \odot w \odot Q$ is an optimal layout for D.

C.2: If (in G) v, w are two vertices of degree two with common neighbors x and y, then $D^* = D/vx$.

To compute an optimal layout L for D from an optimal layout L^* for D^*, it is sufficient to "insert" v in L^* before the leftmost of x_{vx} and y. For instance, if $L^* = P \odot x_{vx} \odot Q \odot y \odot R$ then $L = P \odot v \odot x \odot Q \odot y \odot R$.

Note that, each time that one of the above rules is applied, the number of nodes decreases by one and therefore, the worst case time-complexity is divided by n. When no reduction rules can be applied anymore, the *Branch & Bound* algorithm, that we call $B\&B$, is applied as explained in Section 3.2, and afterwards, the layout for the original graph can be deduced in linear time as explained above.

3.2 Branch & Bound phase

We now describe our *Branch & Bound* algorithm, called $B\&B$. The main contribution of our work consists of the way we cut the exploration of all layouts of V. Intuitively, let $S \subset V$ and P be a layout of S that Procedure $B\&B$ is testing, i.e., let us consider a step when $B\&B$ is considering all layouts of V with prefix P. Moreover, let L_{UB} be the best layout of D obtained so far by $B\&B$. We use the two following pruning rules:

1. Decide, using Lemma 7, if it is useful to explore layouts with prefix P. To this end, we maintain a table $\mathcal{P}$ that contains, among others, some subsets of V. If there is an entry in $\mathcal{P}$ for $S \subset V$ that satisfies some properties, we can decide that the best layout starting with the nodes in S cannot be better than L_{UB} and therefore, we do not test further layouts with prefix P.

2. Greedily extend the current prefix P with a vertex $v \in V \setminus V(P)$ if v satisfies the conditions of Lemma 6. This allows us to restrict the exploration of all the layouts with prefix P to the layouts with prefix $P \odot v$.

As shown in the next section, these two pruning rules allow our algorithm to achieve better performances than existing ones for computing the vertex-separation of digraphs. Let us now describe the recursive procedure $B\&B$ (Algorithm 3) more formally and prove its correctness. It takes as inputs:

- $D = (V, A)$, the considered digraph.
- L_{UB} and UB such that L_{UB} is the best layout of V obtained so far and $UB = \nu(L_{UB})$. Note that UB is an upper bound for $vs(D)$.
- A layout P of a subset $S \subseteq V$. Intuitively, P is the prefix of the layouts that will be tested by this execution of $B\&B$. That is, either Procedure $B\&B$ will find a layout $L = P \odot Q$ of V such that $\nu(L) < \nu(L_{UB})$ or it decides that $\nu(L) \geq \nu(L_{UB})$ for any $L \in \mathcal{L}_P(V)$.
- $\mathcal{P}$ is a set of triples $(S_i, \nu_i, b_i)_{i \leq |\mathcal{P}|}$, where $S_i \subset V$, ν_i is an integer, and b_i is a boolean for any $i \leq |\mathcal{P}|$. Intuitively, $(S_i, \nu_i, b_i) \in \mathcal{P}$ means that a layout P_i of S_i has already been checked, and, if $b_i = 1$, then it is useless to test any other layout of V starting with the nodes in S_i.

  Moreover, $\nu_i = \nu(P_i)$ and $b_i = 0$ if $\nu_i = \min_{L \in \mathcal{L}_{P_i}(V)} \nu(L)$.

The initial values for the inputs of $B\&B$ are: $P = \emptyset$, L_{UB} is any layout L of V and $UB = \nu(L) < |V|$, and $\mathcal{P} = \emptyset$. Let $\mathcal{P} = (S_i, \nu_i, b_i)_{i \leq |\mathcal{P}|}$ and (L_{UB}, UB) be the current values of the global variables at some step of the execution of the algorithm. Then, executing $B\&B(D, P, UB, L_{UB}, \mathcal{P})$, with $P = (v_1, \cdots, v_k)$ being a layout of $S = \{v_1, \cdots, v_k\}$, proceeds as follows.

- If $\nu(P) \geq UB$, then $B\&B(D, P, UB, L_{UB}, \mathcal{P})$ does nothing, i.e., the exploration of the layouts starting by P stops. Indeed, $\min_{L \in \mathcal{L}_P(V)} \nu(L) \geq \nu(P)$ by definition. Therefore, $\min_{L \in \mathcal{L}_P(V)} \nu(L) \geq UB$ and no layout of V starting with P can be better than L_{UB}.
- Else, if (S, x, b) belongs to $\mathcal{P}$, then this means that a layout P' of S has been already tested, $x = \nu(P')$ and $UB \leq \min_{L \in \mathcal{L}_{P'}(V)} \nu(L)$. If $b = 1$ or $\nu(P) \geq \nu(P')$, then $B\&B(D, P, UB, L_{UB}, \mathcal{P})$ does nothing, i.e., the exploration of the layouts starting by P stops. Indeed, either $x < \min_{L \in \mathcal{L}_{P'}(V)} \nu(L)$ (if $b = 1$) or $\nu(P) \geq \nu(P')$. By Lemma 7, $\min_{L \in \mathcal{L}_{P'}(V)} \nu(L) \leq \min_{L \in \mathcal{L}_P(V)} \nu(L)$. Again, no layout of V starting with P can be better than L_{UB}.
- Otherwise, $B\&B(D, P, UB, L_{UB}, \mathcal{P})$ applies the sub-procedure $Greedy(D, P)$ (Algorithm 1) that returns a prefix P' extending P (i.e., P is a prefix of P') and $\min_{L \in \mathcal{L}_P(V)} \nu(L) = \min_{L \in \mathcal{L}_{P'}(V)} \nu(L)$. Then, it calls $B\&B(D, P' \odot v, UB, L_{UB}, \mathcal{P})$ for all $v \in V \setminus V(P')$ such that $\nu(P' \odot v) < UB$, starting from the most promising vertices (i.e., by increasing value of $\nu(P' \odot v)$).

  At the end of the exploration of the layouts with prefix P, we update the table $\mathcal{P}$ (Algorithm 2). That is,

  - if there was no triple $(S, *, *)$ in $\mathcal{P}$, then, $(S, \nu(P), b)$ is added to $\mathcal{P}$ with $b = 0$ if and only if $\nu(P) = \nu(L_{UB})$.
  - else, i.e., if there is $(S, x, 0) \in \mathcal{P}$ with $\nu(P) < x$ and $\nu(P) < UB$, then $(S, \nu(P), b)$ replaces $(S, x, 0)$ in $\mathcal{P}$ with $b = 0$ if and only if $\nu(P) = \nu(L_{UB})$.

Algorithm 1 *Greedy*(D, P).

Require: A digraph $D = (V, A)$, a layout P of $S \subseteq V$

$S := V(P)$ and $P' := P$

while $\exists\, v \in V \setminus S$ s.t. $N^+(v) \subseteq S \cup N^+(S)$ or $\exists\, v \in N^+(S)$ s.t. $|N^+(v) \setminus (S \cup N^+(S))| = 1$ **do**

  $P' := P \odot v$ and $S := S \cup \{v\}$

return P'

Algorithm 2 Update-prefix-table$(D, \mathcal{P}, P, Current, vs^*)$.

if $vs^* < Current$ and $\nu(P) = vs^*$ **then** $b := 0$ **else** $b := 1$

if $(V(P), \nu(P), 0) \in \mathcal{P}$ **then**

  Replace $(V(P), \nu(P), 0)$ with $(V(P), \nu(P), b)$ in $\mathcal{P}$

else

  Add $(V(P), \nu(P), b)$ in $\mathcal{P}$

4 Simulations and interpretation of results

In this section, we evaluate the performance of our algorithm. In particular, we compare it to other exact algorithms and heuristics of the literature. We also analyze the impact of the three optimization phases of our algorithm : pre-processing, greedy steps, and pruning using prefixes. It appears that our algorithm is able to compute the vertex-separation of all graphs with at most 60 nodes but also for some graphs up to 250 nodes. Not only does our algorithm outperforms existing exact algorithms but it can also be used as a good heuristic to obtain upper bounds on the vertex-separation.

4.1 Implementation

We have implemented several variants of our Branch and Bound algorithm in order to analyze the impact of each of our optimization sub-procedures. Our algorithms are implemented in Cython using the *Sagemath* open-source mathematical software [58]. All computations have been performed on a computer equipped with a Intel Xeon CPU operating at 3.20GHz and 64GB of RAM. Note that, all the algorithms take a digraph as input. If the input is a non-directed graph, we consider it as a symmetric directed graph. We now present our variants of the Branch and Bound algorithm.

4.1.1 Our variants of the Branch and Bound algorithm

- **BAB**. This is the basic version of the branch and bound algorithm (as already implemented in [57]). It takes digraph D as an input and is applied sequentially on each of the strongly connected components of D. For each strongly connected component C of D, **BAB** considers all possible layouts of $V(C)$ and keeps a layout L minimizing $\nu(L)$.
- **BAB-P**. In this variant, we add the pre-processing phase to the basic variant **BAB**. That is, if some arc-contractions or node-deletions can be performed on the input digraph, we execute them.
- **BAB-GP**. In this variant, we add the greedy-step process (Algorithm 1) to the variant **BAB-P**.
- **BAB-GPP**. This last variant is our main algorithm which is obtained from **BAB-GP** by adding a cut-process by storing some subsets of nodes a layout of which has already been tested as prefix (Algorithm 2).

For the different variants, it is critical to have a fast access to the data structure representing the digraph. To speed-up critical operations on neighborhoods (union, intersection, size, etc.), we store out-neighborhoods using bitsets (in particular, a single integer when $n \leq 64$). This enables us to use bitwise operations (OR for union, AND for intersection, etc.).

For **BAB-GPP**, the prefixes are stored in a tree structure which allows us a fast access to already tested prefixes. We parameterized the maximum length of a prefix (which corresponds to the tree depth) and set it by default to $\min\{n/3,\ 50\}$. The total number of stored prefixes is also parameterized to limit memory usage. By default we store at most 10^7 prefixes.

Algorithm 3 $B\&B(D, P, vs^*, L^*, \mathcal{P})$.

if $\nu(P) < vs^*$ and $(V(P), \nu(P), 1) \notin \mathcal{P}$ **then**
  $P' := Greedy(D, P)$
  if $V(P') = V$ and $\nu(P') < vs^*$ **then**
    return $(\nu(P'), P')$
  else
    $Current := vs^*$
    for all $v \in V \setminus P'$, by increasing values of $\nu(P' \odot v)$ **do**
      if $\nu(P' \odot v) < vs^*$ **then**
        $(vs'', L'') := B\&B(D, P' \odot v, vs^*, L^*, \mathcal{P})$
        if $vs'' < vs^*$ **then**
          $(vs^*, L^*) := (vs'', L'')$
    Update-prefix-table$(D, \mathcal{P}, P, Current, vs^*)$
return (vs^*, L^*)

Finally, we used a timer to limit computation time per (di)graph. When the time is up, we return the best solution found so far and record it as an upper-bound on the vertex-separation of the input digraph. Following [5], we set this limit to 10 min as both CPUs used are comparable.

4.1.2 Previous exact algorithms.

The pathwidth problem is known to be Fixed Parameter Tractable, that is, $vs(G) \leq k$ can be decided in time $f(k)|V(G)|$ [16]. However, the function f is of order $O(2^{k^c})$ for some $c \geq 2$ [8]. Moreover, pathwidth is unlikely to have a polynomial kernel [10], more precisely, pathwidth and treewidth do not have a polynomial kernel unless $NP \subseteq coNP/poly$ [29]. Few FPT algorithms have been implemented [51] and people have focused on exact exponential-time algorithms.

In addition to the basic Branch and Bound **BAB** of Solano [57], we are aware of the following implementations of algorithms to compute vertex-separation of graphs.

- **MILP.** Some mixed integer linear programming formulations for the vertex-separation (MILP) have been proposed in [5, 57]. A similar formulation has been implemented in *Sagemath*[3]. We use this function for purpose of comparison.
- **DYNPROG.** In [12], a dynamic programming algorithm, called DYNPROG, is designed to compute the vertex-separation of a digraph D. Roughly, this algorithm aims at deciding whether $vs(D) \leq k$ for all $k = 1, \cdots, |V(D)|$. For a given $k \leq |V(D)|$, DYNPROG stores the subsets of $V(D)$ with border at most k and uses dynamic programming to limit the number of stored subsets. An implementation of DYNPROG is also available in *Sagemath* [58][3]. This implementation encodes each subset L and neighborhoods as words of n bits and stores the values $\nu(L)$ in an array of size 2^n. While enabling fast bitwise operations, these implementation choices restrict its usage to graphs with strictly less than 32 nodes.
- **SAT.** In [5], an algorithm to compute the pathwidth of graphs is designed by formulating the PATHWIDTH problem as a Boolean satisfiability testing (SAT) instance and solving this SAT instance using the *MiniSat* solver [45]. No implementations are provided but the algorithm has been tested on the *Rome graphs* dataset [52]. The SAT algorithm have been able to compute the pathwidth of 17% of the instances in the *Rome graphs* dataset.

4.2 Evaluation in random digraphs

In this section, we use the available implementations of MILP and DYNPROG in *Sagemath* to compare their performance with our algorithms in random directed graphs. Random directed graphs with N nodes are generated using the `graphs.RandomDirectedGNM(N,M)` method of *Sagemath* to generate them, where $M = \text{density} * N(N-1)$ is the number of arcs. Recall that in the $G(N, M)$ model, a graph is

[3] Module `sage.graphs.graph_decompositions.vertex_separation`.

chosen uniformly at random among all graphs with N nodes and M arcs [31]. For several network sizes ($N \in \{20, 25, 30, 40, 50\}$), we execute the algorithms for various densities ($0 < density < 1$). For any N and each density, we run the algorithms on 1 000 instances. The same instances are used for all algorithms. The average running times are depicted on Fig. 2.

It took us one week of computations to generate Fig. 2(a) whose purpose is only to show that MILP is not competitive w.r.t. other methods by orders of magnitude. Concerning DYNPROG, it is faster than all other methods when $N = 20$ (Fig. 2(a)), but starting from $N = 25$ (Figs. 2(b) and 2(c)), our algorithms are significantly faster. Recall that we have performed our experiments using the implementation of DYNPROG that is available in [58], which can be used only for graphs with strictly less than 32 nodes.

For $N \geq 30$, the basic variant of **BAB** behaves as well as other variants for $density \geq 0.5$, but for smaller densities the computation time is not competitive (several hours). Therefore, we did not report on its performances on the plots.

Impact of optimization Phases. In Figs. 2(b) to 2(e), we observe that for densities ≥ 0.4, all variants of the Branch and Bound algorithm are very fast with negligeable differences (< 0.01 sec.). However, we observe large variations for smaller densities. More precisely, we observe in Fig. 2(b) the speed-up offered by the pre-processing phases w.r.t. the basic Branch and Bound proposed in [57]. Indeed, removing one vertex of the input digraph reduces the search space by a factor n, and so the overall computation time. We then observe that the Greedy steps offer significant additional speed-up. This is particularly impressive in Fig. 2(c) which reports a reduction of the overall computation time by two orders of magnitude. Finally, Fig. 2(d) reports large speed-up when using prefixes to cut the search space. In Fig. 2(e), we point out that the running time of **BAB-GPP** varies a lot depending on the graphs (especially for graphs with small density). We report on the variations in running time when $N = 50$ and using the best settings of our algorithm. For densities below than 0.4, the running time varies by up to two orders of magnitude, while for larger densities, the range of variations is very small. Last, we have reported in Fig. 2(f) the evolution of the running time of **BAB-GPP** for different densities. This confirms that the higher the density, the larger the size of the graphs we are able to solve. Note that we have observed the same behaviors for all algorithms with undirected graphs.

4.3 Impact of prefix length

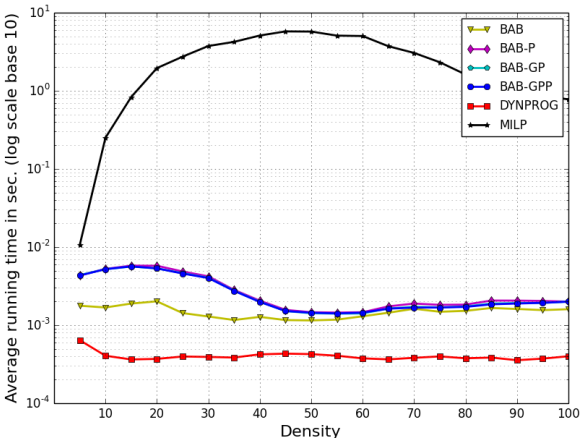
Our main algorithm **BAB-GPP** is parameterized by the maximum length of the prefixes that we store and that allow us to cut the search space during the Branch and Bound process. We have evaluated the impact of this parameter when our algorithm is executed on some specific graphs. In particular, the *Mycielski graph* is considered as a hard instance for integer programming formulations for pathwidth [46]. The Mycielski graph M_k is a triangle-free graph with chromatic number k having the smallest possible number of vertices. Note that, $|V(M_2)| = 2$ and $|V(M_k)| = 2|V(M_{k-1})| + 1$ for any $k > 2$. We ran our algorithm on the Mycielski graph M_k for $k \leq 8$. Our results are depicted in Tab. 1 (for $k \in \{8\}$ we only got upper bound).

k	1	2	3	4	5	6	7	8
$\|V(M_k)\|$	1	2	5	11	23	47	95	191
$pw(M_k)$	0	1	2	5	10	20	38	≤ 72

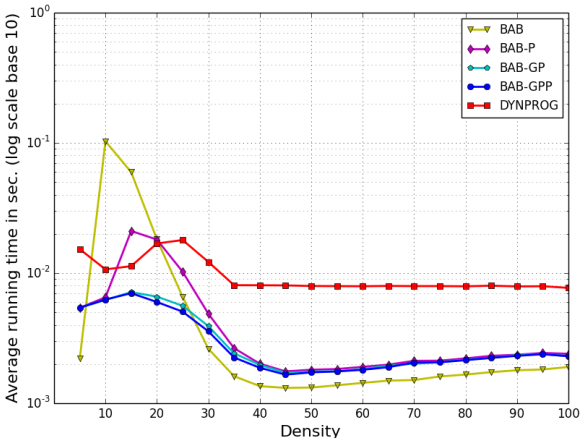
Table 1: Pathwidth of some Mycielski graphs.

Tab. 2 presents the running time of **BAB-GPP** for the Mycielski graph M_6 and different bounds on the size of the stored prefixes. In Tab. 2, the column *Visited nodes* refers to the number of nodes of the Branch and Bound exploration space that are actually considered. Tab. 2 shows that storing larger prefixes allows for an impressive reduction of the computation time. However, we have no improvement here when allowing length of 20 instead of 15, and larger values behave similarly. This is probably due to the fact that in this experiment, because of the greedy steps, the algorithm stores only one prefix of length > 15.

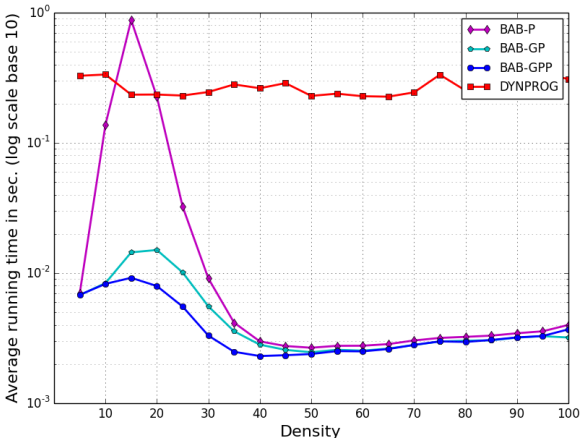
This suggests that the combination of the greedy steps and of the storage of the prefixes allows good performance even with bounded length of the prefixes. Thus, memory-space seems not to be an issue for



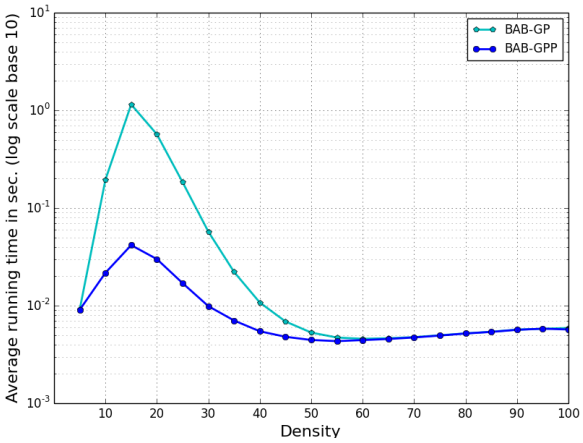
(a) $N = 20$



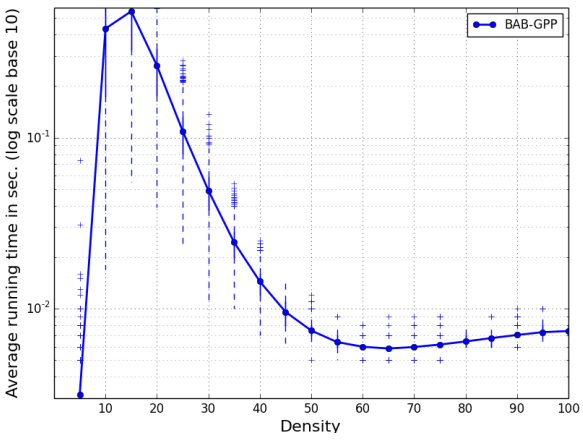
(b) $N = 25$



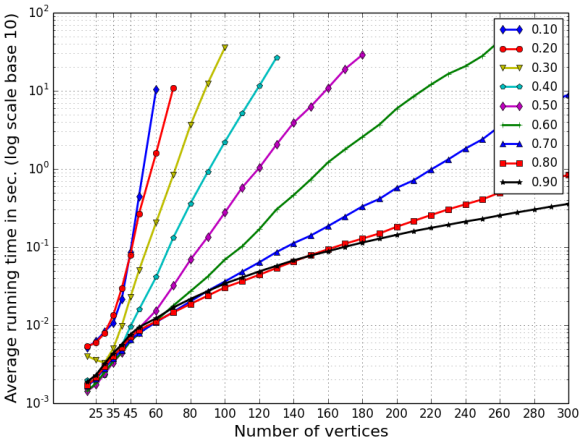
(c) $N = 30$



(d) $N = 40$



(e) $N = 50$, variations



(f) Computation time per densities, using **BAB-GPP**.

Figure 2: Average running time of the algorithms on random digraphs.

Max. prefix length	Time (in sec.)	Visited nodes	Stored prefix
1	2226.45	1 256 780 074	44
5	144.93	82 417 700	16 386
8	5.81	3 319 482	47 756
10	1.07	664 677	61 466
15	0.77	496 482	65 252
20	0.78	496 482	65 253

Table 2: Running time of **BAB-GPP** for the Mycielski graph M_6.

graphs of reasonable size (even though the algorithm might potentially store an exponential number of prefixes).

This observation is further supported by experiments on the Mycielsky graph M_7 with 95 nodes. While the algorithm was not able to return a solution after one month of computation when restricting the size of the prefixes to 20, we got the optimal solution in 14 minutes storing 10 496 595 prefixes of size up to 30, and in 10 minutes storing 10 497 628 of size up to 40. Larger values are not helpful here.

4.4 Comparison of BAB-GPP with SAT on *Rome graphs*

The *Rome graphs* dataset [52] consists of 11 534 undirected n-node graphs with $10 \leq n \leq 100$. In [5], the performance of the SAT algorithm has been evaluated using this benchmark[4]. With 10 minutes time limit per graph, the algorithm of [5] has been able to compute the pathwidth for 17.0% of the Rome graphs. In particular, it is stated that *"We note that almost all small graphs ($n + m < 45$) could be solved within the given timeout, however, for larger graphs, the percentage of solved instances rapidly drops [...] Almost no graphs with $n + m > 70$ were solved."* [5] (where m is the number of edges of the considered graphs).

In contrast, our algorithm has computed the pathwidth of 95.6% of the graphs in the *Rome dataset*, with same time limit of 10 minutes Note that, in particular, we solved all instances with at most 82 vertices. We report in Tab. 3 the repartition of (un)solved instances.

N	≤ 82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	**Total**
Nb graphs	9586	86	78	73	70	68	69	63	59	116	119	134	154	139	148	139	143	144	141	11 529
Solved	9586	85	77	68	65	62	65	59	50	96	91	104	113	91	110	80	77	70	78	11 027
Unsolved	0	1	1	5	5	6	4	4	9	20	28	30	41	48	38	59	66	74	63	502

Table 3: Repartition of (un)solved instances.

4.5 TreewidthLIB

The TreewidthLIB [62] contains 710 graphs among which 387 are obtained from others by pre-processing. We considered only the 322 non pre-processed graphs. We were able to solve 139 of these graphs (in less than a second for most of them). We report some of these results in Tab. 4. Furthermore, we were able to solve 27 out of the 62 Delaunay triangulation of TSP instances (see Tab. 5). Observe that the pre-processing phase has not been able to reduce the size of any of these instances.

4.6 Using BAB-GPP as a heuristic

Some research has been devoted to design heuristic algorithms for this problem. [30, 53] proposed the following: starting from some layout of the nodes of D, the solution is improved by switching pairs of

[4]Computations in [5] have been performed on a computer equipped with a AMD Opteron 6172 processor operating at 2.1GHz and 256GB of RAM.

Name	N	M	N^*	M^*	**pw**	**time (in sec.)**
BN_28	24	49	23	48	5	0.004
BN_29	24	49	23	48	5	0.004
alarm	37	65	36	64	4	0.006
barley	48	126	–	–	7	0.006
david	87	406	78	397	13	99.47
diabetes	413	819	–	–	6	122.6
fungiuk	15	36	–	–	4	0.001
games120	120	638	–	–	32	148.4
graph03	100	340	–	–	21	19.18
graph05	100	416	–	–	25	65.0
knights8_8	64	168	–	–	16	7.246
mainuk	48	84	47	83	6	0.012
mildew	35	80	–	–	5	0.002
miles750	128	2 113	–	–	36	51.19
miles1000	128	3 216	–	–	49	0.696
miles1500	128	5 198	–	–	77	0.098
mulsol.i.5	176	3 973	–	–	31	67.31
oesoca	39	67	35	63	4	0.007
oesoca42	42	72	36	66	4	0.007
oesoca+	67	208	60	201	11	54.73
queen5_5	25	160	–	–	18	0.004
queen6_6	36	290	–	–	25	0.008
queen7_7	49	476	–	–	35	0.022
queen8_8	64	728	–	–	45	0.093
queen8_12	96	1 368	–	–	65	4.262
queen9_9	81	1 056	–	–	58	1.851
queen10_10	100	1 470	–	–	72	25.92
sodoku	81	810	–	–	45	2.569
sodoku-elim1	80	898	–	–	45	1.107
water	32	123	–	–	10	0.004
weeduk	15	24	12	21	4	0.002
zeroin.i.1	126	4 100	–	–	50	0.937

Table 4: Some computational results on solved instances from [62].

nodes in the layout until no further improvement can be obtained. The VSPLIB [63] has been designed for benchmarking this heuristic [30]. VSPLIB [63] contains 173 instances: 50 $\gamma \times \gamma$ grids with $5 \leq \gamma \leq 54$; 50 trees with respectively 22, 67, and 202 nodes and pathwidth 3, 4, and 5; a set of 73 graphs of order n, called HB, with $10 \leq n \leq 960$ (see [30] for more details). In particular, the pathwidth of $\gamma \times \gamma$ grids equals γ and the pathwidth of trees can be computed in linear time [55]. Therefore, this benchmark allows to evaluate the performance of heuristic. We tested the Algorithm **BAB-GPP** as a heuristic on the graphs of VSPLIB. If the time is up before the end of **BAB-GPP**'s execution, the algorithm returns the value computed so far (i.e., an upper bound on the pathwidth of the graph).

We were able to compute the exact pathwidth for all grids with side $\gamma \leq 13$, trees with $n \leq 67$, and 26 of the HB graphs including one with 957 nodes. For all grids and trees of the VSPLIB, the final value returned by **BAB-GPP** equals the exact pathwidth. That is, our algorithm always finds quickly an optimal layout and most of the execution time is devoted to prove its optimality.

In particular for a $\gamma \times \gamma$ grid, the first solution found is always the pathwidth. This is due to the order in which we add the vertices in the layouts: starting from a node with smallest degree and always adding a node that minimizes the increase of the size of the border. Proceeding that way will always give an optimal layout in square grids.

For trees, the first layout tested by **BAB-GPP** is not always the optimal one, but an optimal one is found very quickly. It would be interesting to understand why our Branch and Bound algorithm performs well in trees[5].

[5]Note that optimal path-decompositions are computable in linear time in trees [55].

Name	N	M	pw	time (in sec.)
bier127.tsp	127	368	15	2.797
ch130.tsp	130	377	12	0.358
ch150.tsp	150	432	13	0.788
eil101.tsp	101	290	11	0.045
eil51.tsp	51	140	8	0.005
eil76.tsp	76	215	11	0.027
kroA100.tsp	100	285	10	0.025
kroA150.tsp	150	432	12	1.189
kroA200.tsp	200	586	13	1.266
kroB100.tsp	100	284	10	0.023
kroB150.tsp	150	436	12	0.4
kroB200.tsp	200	580	13	1.511
kroC100.tsp	100	286	10	0.025
kroE100.tsp	100	283	9	0.015
lin105.tsp	105	292	9	0.015
pr76.tsp	76	218	10	0.013
pr107.tsp	107	283	7	0.01
pr124.tsp	124	318	10	3.781
pr136.tsp	136	377	10	0.065
pr144.tsp	144	393	10	0.124
pr152.tsp	152	428	11	0.405
pr226.tsp	226	586	8	6.139
rat195.tsp	195	562	13	1.247
rat99.tsp	99	279	10	0.023
rd100.tsp	100	286	11	0.043
tsp225.tsp	225	622	13	116.8
u159.tsp	159	431	12	1.912

Table 5: Computational results for Delaunay triangulation of TSP instances from [62].

5 Conclusion

In this paper, we have presented a new Branch and Bound algorithm for computing the pathwidth of graphs and the vertex-separation of digraphs. Our approach is more promising than previous proposals, based on ILP or SAT, for solving large instances. Indeed, the drawbacks of ILP and SAT formulations for layout problems are both in the large number of symmetries of the problems, and in the time needed to fill the optimality gap (i.e., distance between the lower bound based on the fractional relaxation of the formulation and the best integral solution).

Very recently, another very similar branch and bound algorithm has been proposed to compute the pathwidth of graphs [43]. Their algorithm has been tested on TreewidthLIB and it appears that, in most of the cases, our algorithm is better. Working on the instances for which their algorithm is better leads us to the following observation: the ordering in which the vertices are initially labeled (i.e., their names in the encoding of the graph) has a strong impact on the performance of our algorithm. Indeed, when extending a prefix, our algorithm tries all possible nodes in some ordering ("by increasing value of $\nu(P' \odot v)$"). However, ties are broken using the identifiers of the vertices. Our recent experiments have shown that, on the instances on which the algorithm of [43] seemed better, just reordering randomly the identifiers of the nodes allowed our algorithm to achieve same performances as [43]. Hence, one important next step to improve our algorithm will be to study in more detail the impact of the initial labeling of the nodes, as well as alternative tie-breaking methods (i.e., methods to order vertices with same value $\nu(P' \odot v)$ Line 7 of Algorithm 3).

On the other hand, we will search for new pruning rules to further reduce computation time. In particular, looking for good lower bounds for pathwidth of graphs is a theoretical issue that has not received much attention. It would moreover speed up our algorithm since the main computation time seems dedicated to prove the optimality of the best value computed. In particular, lower bounds for pathwidth that are not lower bounds for treewidth would be very interesting. It would also be interesting to find new pre-processing rules to further reduce the size of the input digraph. For instance, does Rule

C.2 extend to directed graphs?

Finally, the branch-and-bound algorithm presented in this paper (without the pre-processing phase) is now included in *Sagemath* and so it can be used by others to perform comparisons on a fair basis.

References

[1] E. H. Bachoore and H. L. Bodlaender. New upper bound heuristics for treewidth. In *4th InternationalWorkshop on Experimental and Efficient Algorithms (WEA)*, volume 3503 of *LNCS*, pages 216–227. Springer, 2005.

[2] E. H. Bachoore and H. L. Bodlaender. A branch and bound algorithm for exact, upper, and lower bounds on treewidth. In *Second International Conference on Algorithmic Aspects in Information and Management, AAIM*, volume 4041 of *LNCS*, pages 255–266. Springer, 2006.

[3] J. Barát. Directed path-width and monotonicity in digraph searching. *Graphs and Combinatorics*, 22(2):161–172, 2006.

[4] D. Berwanger, A. Dawar, P. Hunter, S. Kreutzer, and J. Obdrzálek. The dag-width of directed graphs. *J. Comb. Theory, Ser. B*, 102(4):900–923, 2012.

[5] T. C. Biedl, T. Bläsius, B. Niedermann, M. Nöllenburg, R. Prutkin, and I. Rutter. Using ILP/SAT to determine pathwidth, visibility representations, and other grid-based graph drawings. In *Graph Drawing*, volume 8242 of *LNCS*, pages 460–471. Springer, 2013.

[6] D. Bienstock. Graph searching, path-width, tree-width and related problems (a survey). *DIMACS Ser. in Discrete Mathematics and Theoretical Computer Science*, 5:33–49, 1991.

[7] D. Bienstock and P. D. Seymour. Monotonicity in graph searching. *J. Algorithms*, 12(2):239–245, 1991.

[8] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[9] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998.

[10] H. L. Bodlaender, R. G. Downey, M. R. Fellows, and D. Hermelin. On problems without polynomial kernels. *J. Comput. Syst. Sci.*, 75(8):423–434, 2009.

[11] H. L. Bodlaender, P. G. Drange, M. S. Dregi, F. V. Fomin, D. Lokshtanov, and M. Pilipczuk. An $o(c^k n)$ 5-approximation algorithm for treewidth. In *54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 499–508. IEEE Computer Society, 2013.

[12] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos. A note on exact algorithms for vertex ordering problems on graphs. *Theory Comput. Syst.*, 50(3):420–432, 2012.

[13] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos. On exact algorithms for treewidth. *ACM Transactions on Algorithms*, 9(1):12, 2012.

[14] H. L. Bodlaender, J. Gustedt, and J. A. Telle. Linear-time register allocation for a fixed number of registers. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 574–583. ACM/SIAM, 1998.

[15] H. L. Bodlaender, B. M. P. Jansen, and S. Kratsch. Kernel bounds for structural parameterizations of pathwidth. In *13th Scandinavian Symp. and Workshops on Algorithm Theory (SWAT)*, volume 7357 of *LNCS*, pages 352–363. Springer, 2012.

[16] H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*, 21(2):358–402, 1996.

[17] H. L. Bodlaender, T. Kloks, and D. Kratsch. Treewidth and pathwidth of permutation graphs. *SIAM J. Discrete Math.*, 8(4):606–616, 1995.

[18] H. L. Bodlaender and A. M. C. A. Koster. On the maximum cardinality search lower bound for treewidth. *Discrete Applied Mathematics*, 155(11):1348–1372, 2007.

[19] H. L. Bodlaender and A. M. C. A. Koster. Treewidth computations I. upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[20] H. L. Bodlaender and A. M. C. A. Koster. Treewidth computations II. lower bounds. *Inf. Comput.*, 209(7):1103–1119, 2011.

[21] H. L. Bodlaender and R. H. Möhring. The pathwidth and treewidth of cographs. *SIAM J. Discrete Math.*, 6(2):181–188, 1993.

[22] N. Cohen, D. Coudert, D. Mazauric, N. Nepomuceno, and N. Nisse. Tradeoffs in process strategy games with application in the WDM reconfiguration problem. *Theo. Comp. Sci.*, 412(35):4675–4687, 2011.

[23] D. Coudert, F. Huc, D. Mazauric, N. Nisse, and J.-S. Sereni. Reconfiguration of the routing in WDM networks with two classes of services. In *Optical Network Design and Modeling (ONDM)*, pages 1–6. IEEE, 2009.

[24] D. Coudert, D. Mazauric, and N. Nisse. Experimental evaluation of a branch and bound algorithm for computing pathwidth. In *13th International Symposium on Experimental Algorithms (SEA)*, volume 8504 of *LNCS*, pages 46–58. Springer, 2014.

[25] D. Coudert and J.-S. Sereni. Characterization of graphs and digraphs with small process number. *Discr. Appl. Math.*, 159(11):1094–1109, 2011.

[26] B. Courcelle and M. Mosbah. Monadic second-order evaluations on tree-decomposable graphs. *Theor. Comput. Sci.*, 109(1&2):49–82, 1993.

[27] N. Deo, S. Krishnamoorthy, and M. A. Langston. Exact and approximate solutions for the gate matrix layout problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6:79–84, 1987.

[28] J. Díaz, J. Petit, and M. Serna. A survey on graph layout problems. *ACM Comput. Surveys*, 34(3):313–356, 2002.

[29] A. Drucker. New limits to classical and quantum instance compression. In *53rd Annual IEEE Symposium on Foundations of Computer Science, (FOCS)*, pages 609–618. IEEE Computer Society, 2012.

[30] A. Duarte, L. F. Escudero, R. Martí, N. Mladenovic, J. J. Pantrigo, and J. Sánchez-Oro. Variable neighborhood search for the vertex separation problem. *Computers & OR*, 39(12):3247–3255, 2012.

[31] P. Erdös and A. Rényi. On random graphs. I. *Publicationes Mathematicae*, 6:290297, 1959.

[32] U. Feige, M. T. Hajiaghayi, and J. R. Lee. Improved approximation algorithms for minimum-weight vertex separators. In *37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 563–572. ACM, 2005.

[33] M. R. Fellows and M. A. Langston. On well-partial-order theory and its application to combinatorial problems of VLSI design. *SIAM J. Discrete Math.*, 5(1):117–126, 1992.

[34] V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI'04, pages 201–208, Arlington, Virginia, United States, 2004. AUAI Press.

[35] J. Gustedt. On the pathwidth of chordal graphs. *Discrete Applied Maths*, 45(3):233–248, 1993.

[36] A. Hein and A. M. C. A. Koster. An experimental evaluation of treewidth at most four reductions. In *10th Int. Symposium on Experimental Algorithms (SEA)*, volume 6630 of *LNCS*, pages 218–229. Springer, 2011.

[37] P. Hunter and S. Kreutzer. Digraph measures: Kelly decompositions, games, and orderings. *Theor. Comput. Sci.*, 399(3):206–219, 2008.

[38] T. Johnson, N. Robertson, P. D. Seymour, and R. Thomas. Directed tree-width. *J. Comb. Theory, Ser. B*, 82(1):138–154, 2001.

[39] N. G. Kinnersley. The vertex separation number of a graph equals its pathwidth. *Inform. Process. Lett.*, 42(6):345–350, 1992.

[40] L. M. Kirousis and C. H. Papadimitriou. Searching and pebbling. *Theor. Comput. Sci.*, 47(3):205–218, 1986.

[41] K. Kitsunai, Y. Kobayashi, K. Komuro, H. Tamaki, and T. Tano. Computing directed pathwidth in $O(1.89^n)$ time. In *International Symposium on Parameterized and Exact Computation (IPEC)*, volume 7535 of *Lecture Notes in Computer Science*, pages 182–193. Springer, 2012.

[42] T. Kloks, H. L. Bodlaender, H. Müller, and D. Kratsch. Computing treewidth and minimum fill-in: All you need are the minimal separators. In *First Annual European Symposium on Algorithms (ESA)*, volume 726 of *LNCS*, pages 260–271. Springer, 1993.

[43] Y. Kobayashi, K. Komuro, and H. Tamaki. Search space reduction through commitments in pathwidth computation: An experimental study. In *13th International Symposium on Experimental Algorithms (SEA)*, volume 8504 of *LNCS*, pages 388–399. Springer, 2014.

[44] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou. The complexity of searching a graph. *J. ACM*, 35(1):18–44, 1988.

[45] MiniSat, a minimalistic open-source SAT solver. `http://minisat.se/`.

[46] MIPLIB - mixed integer problem library. `http://miplib.zib.de/`.

[47] B. Monien and I. H. Sudborough. Min cut is np-complete for edge weighted trees. *Theor. Comput. Sci.*, 58:209–229, 1988.

[48] B. A. Reed. Finding approximate separators and computing tree width quickly. In *24th Annual ACM Symposium on Theory of Computing (STOC)*, pages 221–228. ACM, 1992.

[49] N. Robertson and P. D. Seymour. Graph minors. I. excluding a forest. *J. Comb. Theory, Ser. B*, 35(1):39–61, 1983.

[50] N. Robertson and P. D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[51] H. Röhrig. Tree decomposition: a feasibility study. `http://hein.roehrig.name/dipl/`, 1998. Diplomarbeit (master's thesis), Universität des Saarlandes.

[52] Rome graphs. `http://www.graphdrawing.org/download/rome-graphml.tgz`.

[53] J. Sánchez-Oro, J. J. Pantrigo, and A. Duarte. Combining intensification and diversification strategies in vns. an application to the vertex separation problem. *Computers & Operations Research*, 52, Part B:209–219, 2014.

[54] P. D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[55] K. Skodinis. Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. *J. Algorithms*, 47(1):40–59, 2003.

[56] F. Solano. Analyzing two different objectives of the WDM network reconfiguration problem. In *Proc. IEEE Globecom*, pages 1–7, 2009.

[57] F. Solano and M. Pióro. Lightpath reconfiguration in WDM networks. *IEEE/OSA J. Opt. Commun. Netw.*, 2(12):1010–1021, 2010.

[58] W. Stein et al. *Sage Mathematical Software System (Version 6.6)*. The Sage Development Team, 2015. `http://www.sagemath.org`.

[59] K. Suchan and I. Todinca. Pathwidth of circular-arc graphs. In *Proc. WG*, pages 258–269, 2007.

[60] K. Suchan and Y. Villanger. Computing pathwidth faster than 2^n. In *4th International Workshop on Parameterized and Exact Computation (IWPEC)*, volume 5917 of *LNCS*, pages 324–335. Springer, 2009.

[61] M. E. R. van Boxel. Improved algorithms for the computation of special junction trees. `http://dspace.library.uu.nl/handle/1874/296605`, july 2014. Master thesis, Utrecht University.

[62] J.-W. van den Broek and H. Bodlaender. TreewidthLIB, a benchmark for algorithms for treewidth and related graph problems. `http://www.cs.uu.nl/research/projects/treewidthlib/`.

[63] VSPLIB. `http://www.optsicom.es/vsp/`, 2012.

[64] B. Yang and Y. Cao. Digraph searching, directed vertex separation and directed pathwidth. *Discrete Applied Mathematics*, 156(10):1822–1837, 2008.

[65] B. Yang and Y. Cao. Monotonicity in digraph search problems. *Theor. Comput. Sci.*, 407(1-3):532–544, 2008.